IMPROVED METHODS AND APPARATUS FOR FAST FOURIER TRANSFORM

The invention provides improved methods and apparatus for fast fourier transform.

From the user's perspective, the code performs an in-place "split-complex" 1D FFT (forward or inverse) for power of 2 sizes ranging from 16 to 4096, inclusive.

There are 3 user-callable functions: fft_setup(), fft_z() and fft_free():

```
void fft_setup ( unsigned long LOG2N, FFT_setup *SETUP );
void fft_z ( float *Creal, float *Cimag, unsigned long LOG2N, FFT_setup *SETUP );
void fft_free ( FFT_setup *SETUP );
```

FFT_setup is a structure defined as follows:

```
typedef struct {
  float          *twidp;     /* pointer to 16-byte aligned
malloc'ed twiddle buffer */
  unsigned char  *bitrp;     /* pointer to static bit-reversal
table */
}FFT_setup;
```

A user first calls fft_setup() specifying a particular FFT size (actually, the base 2 log of the size) along with a pointer to an uninitialized FFT_setup structure. This function allocates (malloc) and builds the appropriate "twiddle" table and places a pointer to this table and the appropriate bit-reversal table (a static table) in the FFT_setup structure supplied by the caller.

Next, fft_z() can be called repeatedly for the same size FFT as was specified in the corresponding call to fft_setup(). The user must also specify the same FFT_setup structure that was filled in by that call. The input/output vectors are supplied in a split-complex format with the real parts contiguous in the first float vector argument (Creal) and the corresponding imaginary parts contiguous in the second float vector argument (Cimag). The call performs a forward FFT. To perform an inverse FFT, simply interchange the real and imaginary vectors (i.e., specify the imaginary vector in the first argument and the real vector in the second argument).

Finally, the user calls fft_free() to free the twiddle buffer previously allocated and constructed by fft_setup(). The user must specify the same FFT_setup structure to both calls.

Here is a one line description of what is in each file:

fft.h:	user's header file
fft_bitr:	contains static bit-reversal tables for all 9 FFT sizes (16 - 4096)
fft_setup.c	source for fft_setup() and fft_free()
fft_z.c	source for fft_z()
ppc_vmx.h:	macro header file for VMX (altivec) emulation of SIMD instructions.

ppc_vmx.c: contains C functions that emulate VMX (altivec) SIMD instructions

Note that fft_z() is implemented using macros that emulate VMX SIMD instructions. There is a structure (VMX_reg) defined in ppc_vmx.h that emulates a 16-byte VMX SIMD register. The floating point variables used in fft_z() are of this type. fft_z.c does *not* contain an optimized PPC G4 implementation of fft_z() insofar as the instructions are *not* ordered in an optimal way for that processor. However, the primary patent claim is clearly demonstrated in the final pass of the FFT which begins on line 661 of fft_z.c. This section performs the final radix-4 in-place pass of the FFT but manages to leave the results correctly ordered in the real and imaginary input/output vectors. This can be accomplished with 32 or fewer 16-byte "registers" (i.e., 512 or fewer bytes of temporary storage).

It will be appreciated that the teachings hereof may be applied using different programming languages, toolsets, operating systems, platforms and otherwise.

```
/*****************************************************************\
|*  File Name:     fft.h                                          *|
|*  Description:   Header file for FFT functions                  *|
|*                                                                *|
|*             Mercury Computer Systems, Inc.                     *|
|*             Copyright (c) 1999  All rights reserved            *|
|*                                                                *|
|* Revision      Date        Engineer; Reason                     *|
|* --------      ----        ----------------                     *|
|*   0.0         991119       jg;  Created                        *|
\*****************************************************************/

/*
 *  FFT setup structure
 *  contains pointers to twiddles and bit-reversed indices
 *  pointers are filled in by fft_setup() function
 */
typedef struct {
   float          *twidp;
   unsigned char  *bitrp;
} FFT_setup;

/*
 *  FFT function prototypes
 */
void  fft_free( FFT_setup *SETUP );
void  fft_setup( unsigned long LOG2N, FFT_setup *SETUP );
void  fft_z( float *Cr, float *Ci, unsigned long LOG2N, FFT_setup *SETUP );
```

```
/*****************************************************************\
|*  File Name:     fft_bitr.c                                      *|
|*  Description:   Special bit-reversed tables for FFT sizes      *|
|*                 4 <= LOG2N <= 12                                *|
|*                                                                 *|
|*    Let:  LOG2M = LOG2N - 4                                      *|
|*          M = 2 ^ LOG2M                                          *|
|*                                                                 *|
|*    For each table:                                              *|
|*                                                                 *|
|*      section 1:                                                 *|
|*        n1 = bitr[0] = # of elements in section 1                *|
|*        (The first and second elements are not in the table      *|
|*        as they are known to be 0 and M-1, respectively.)        *|
|*                                                                 *|
|*        0, M-1, bitr[1], ..., bitr[n1-2] =                       *|
|*        indices that bit-reverse to themselves                   *|
|*                                                                 *|
|*      section 2:                                                 *|
|*        n2 = bitr[n1-1] = # of elements in section 2             *|
|*        It's always true that n1 + n2 = M.                       *|
|*        (The first element is not in the table and, if           *|
|*        n2 != 0, is known to be 1.)                              *|
|*                                                                 *|
|*        (1, bitr[n1]), (bitr[n1+1], bitr[n1+2]), ...,            *|
|*        (bitr[M-3], bitr[M-2]) = n2/2 pairs of indices that      *|
|*        bit-reverse to each other. bitr[M-1] = 0.                *|
|*                                                                 *|
|*            Mercury Computer Systems, Inc.                       *|
|*            Copyright (c) 1996  All rights reserved              *|
|*                                                                 *|
|* Revision      Date        Engineer; Reason                      *|
|* --------      ----        ----------------                      *|
|*   0.0        990716        jg;  Created                         *|
\*****************************************************************/

/*
 *  Table for M = 1 (N = 16).
 */
unsigned char  _fft_bitr_1[] = {
   1,
   0, 0, 0
};

/*
 *  Table for M = 2 (N = 32).
 */
unsigned char  _fft_bitr_2[] = {
   2,
   0, 0, 0
};

/*
 *  Table for M = 4 (N = 64).
 */
unsigned char  _fft_bitr_4[] = {
```

```
    2,
    2, 2, 0
};

/*
 *  Table for M = 8 (N = 128).
 */
unsigned char  _fft_bitr_8[] = {
    4, 2, 5,
    4, 4, 3, 6, 0
};

/*
 *  Table for M = 16 (N = 256).
 */
unsigned char  _fft_bitr_16[] = {
    4, 6, 9,
    12, 8, 2, 4, 3, 12, 5, 10, 7, 14, 11, 13, 0
};

/*
 *  Table for M = 32 (N = 512).
 */
unsigned char  _fft_bitr_32[] = {
    8, 4, 10, 14, 17, 21, 27,
    24, 16, 2, 8, 3, 24, 5, 20, 6, 12, 7, 28,
    9, 18, 11, 26, 13, 22, 15, 30, 19, 25, 23, 29, 0
};

/*
 *  Table for M = 64 (N = 1024).
 */
unsigned char  _fft_bitr_64[] = {
    8, 12, 18, 30, 33, 45, 51,
    56, 32, 2, 16, 3, 48, 4, 8, 5, 40, 6, 24,
    7, 56, 9, 36, 10, 20, 11, 52, 13, 44, 14, 28,
    15, 60, 17, 34, 19, 50, 21, 42, 22, 26, 23, 58,
    25, 38, 27, 54, 29, 46, 31, 62, 35, 49, 37, 41,
    39, 57, 43, 53, 47, 61, 55, 59, 0
};

/*
 *  Table for M = 128 (N = 2048).
 */
unsigned char  _fft_bitr_128[] = {
    16, 8, 20, 28, 34, 42, 54, 62, 65, 73, 85, 93, 99, 107, 119,
    112, 64, 2, 32, 3, 96, 4, 16, 5, 80, 6, 48, 7, 112, 9, 72,
    10, 40, 11, 104, 12, 24, 13, 88, 14, 56, 15, 120, 17, 68, 18, 36,
    19, 100, 21, 84, 22, 52, 23, 116, 25, 76, 26, 44, 27, 108, 29, 92,
    30, 60, 31, 124, 33, 66, 35, 98, 37, 82, 38, 50, 39, 114, 41, 74,
    43, 106, 45, 90, 46, 58, 47, 122, 49, 70, 51, 102, 53, 86, 55, 118,
    57, 78, 59, 110, 61, 94, 63, 126, 67, 97, 69, 81, 71, 113, 75, 105,
    77, 89, 79, 121, 83, 101, 87, 117, 91, 109, 95, 125, 103, 115, 111, 123, 0
};

/*
 *  Table for M = 256 (N = 4096).
```

```
*/
unsigned char  _fft_bitr_256[] = {
   16, 24, 36, 60, 66, 90, 102, 126, 129, 153, 165, 189, 195, 219, 231,
   240, 128, 2, 64, 3, 192, 4, 32, 5, 160, 6, 96, 7, 224, 8, 16,
   9, 144, 10, 80, 11, 208, 12, 48, 13, 176, 14, 112, 15, 240, 17, 136,
   18, 72, 19, 200, 20, 40, 21, 168, 22, 104, 23, 232, 25, 152, 26, 88,
   27, 216, 28, 56, 29, 184, 30, 120, 31, 248, 33, 132, 34, 68, 35, 196,
   37, 164, 38, 100, 39, 228, 41, 148, 42, 84, 43, 212, 44, 52, 45, 180,
   46, 116, 47, 244, 49, 140, 50, 76, 51, 204, 53, 172, 54, 108, 55, 236,
   57, 156, 58, 92, 59, 220, 61, 188, 62, 124, 63, 252, 65, 130, 67, 194,
   69, 162, 70, 98, 71, 226, 73, 146, 74, 82, 75, 210, 77, 178, 78, 114,
   79, 242, 81, 138, 83, 202, 85, 170, 86, 106, 87, 234, 89, 154, 91, 218,
   93, 186, 94, 122, 95, 250, 97, 134, 99, 198, 101, 166, 103, 230, 105, 150,
   107, 214, 109, 182, 110, 118, 111, 246, 113, 142, 115, 206, 117, 174, 119,
238,
   121, 158, 123, 222, 125, 190, 127, 254, 131, 193, 133, 161, 135, 225, 137,
145,
   139, 209, 141, 177, 143, 241, 147, 201, 149, 169, 151, 233, 155, 217, 157,
185,
   159, 249, 163, 197, 167, 229, 171, 213, 173, 181, 175, 245, 179, 205, 183,
237,
   187, 221, 191, 253, 199, 227, 203, 211, 207, 243, 215, 235, 223, 251, 239,
247, 0
};
```

```
/*********************************************************************\
|*  File Name:      fft_setup.c                                      *|
|*  Description:    Setup for fft_z (split complex in-place FFT)     *|
|*  Entry/params:   void fft_setup ( ulong LOG2N,                    *|
|*                                   FFT_setup *SETUP )              *|
|*  Entry/params:   void fft_free ( FFT_setup *SETUP )               *|
|*                                                                   *|
|*  Formula:                                                         *|
|*                                                                   *|
|*    LOG2N is the log (base 2) of the FFT size.                     *|
|*      (4 <= LOG2N <= 12)                                           *|
|*                                                                   *|
|*    Let:  N = 2 ^ LOG2N                                            *|
|*          LOG2M = LOG2N - 4                                        *|
|*          M = 2 ^ LOG2M                                            *|
|*          A = 2 * PI / N                                           *|
|*          BITR( i, m ) = bit-reversal of unsigned integer i        *|
|*                         over m bits                               *|
|*                                                                   *|
|*  void fft_setup ( ulong LOG2N, FFT_setup *SETUP )                 *|
|*                                                                   *|
|*    SETUP->twidp is set to an allocated buffer that is             *|
|*      16-byte aligned and contains M sets of 4 x 4 floating        *|
|*      point twiddles arranged exactly as follows:                  *|
|*                                                                   *|
|*      cos(kA), cos((k+1)A), cos((k+2)A), cos((k+3)A),              *|
|*      sin(kA), sin((k+1)A), sin((k+2)A), sin((k+3)A),              *|
|*      cos(2kA), cos(2(k+1)A), cos(2(k+2)A), cos(2(k+3)A),          *|
|*      sin(2kA), sin(2(k+1)A), sin(2(k+2)A), sin(2(k+3)A)           *|
|*                                                                   *|
|*        for k = 0                                                  *|
|*                                                                   *|
|*      cos(kA), cos((k+1)A), cos((k+2)A), cos((k+3)A),              *|
|*      tan(kA), tan((k+1)A), tan((k+2)A), tan((k+3)A),              *|
|*      cot(2kA), cot(2(k+1)A), cot(2(k+2)A), cot(2(k+3)A),          *|
|*      sin(2kA), sin(2(k+1)A), sin(2(k+2)A), sin(2(k+3)A)           *|
|*                                                                   *|
|*        for k = 4 * BITR( 1, LOG2M ),                              *|
|*                4 * BITR( 2, LOG2M ),                              *|
|*                  ...,                                             *|
|*                4 * BITR( M-2, LOG2M )                             *|
|*                                                                   *|
|*      cos(kA), cos((k+1)A), cos((k+2)A), cos((k+3)A),              *|
|*      sin(kA), sin((k+1)A), sin((k+2)A), sin((k+3)A),              *|
|*      cos(2kA), cos(2(k+1)A), cos(2(k+2)A), cos(2(k+3)A),          *|
|*      sin(2kA), sin(2(k+1)A), sin(2(k+2)A), sin(2(k+3)A)           *|
|*                                                                   *|
|*        for k = 4 * (M - 1)                                        *|
|*                                                                   *|
|*    SETUP->bitrp is set to static table of M unsigned char         *|
|*      bit-reversed index values (LOG2M bits) arranged              *|
|*      as follows:                                                  *|
|*                                                                   *|
|*      section 1:                                                   *|
|*        n1 = bitrp[0] = # of elements in section 1                 *|
|*        (The first and second elements are not in the table        *|
```

```
|*        as they are known to be 0 and M-1, respectively.)      *|
|*                                                               *|
|*        0, M-1, bitrp[1], ..., bitrp[n1-2] =                   *|
|*        indices that bit-reverse to themselves                 *|
|*                                                               *|
|*      section 2:                                               *|
|*        n2 = bitrp[n1-1] = # of elements in section 2          *|
|*        It's always true that n1 + n2 = M.                     *|
|*        (The first element is not in the table and, if         *|
|*        n2 != 0, is known to be 1.)                            *|
|*                                                               *|
|*        (1, bitrp[n1]), (bitrp[n1+1], bitrp[n1+2]), ...,       *|
|*        (bitrp[M-3], bitrp[M-2]) = n2/2 pairs of indices that *|
|*        bit-reverse to each other. bitrp[M-1] = 0.             *|
|*                                                               *|
|*  void fft_free ( FFT_setup *SETUP )                           *|
|*                                                               *|
|*    frees SETUP->twidp and sets SETUP->twidp and               *|
|*       SETUP->bitrp to 0                                       *|
|*                                                               *|
|*           Mercury Computer Systems, Inc.                      *|
|*           Copyright (c) 1999  All rights reserved             *|
|*                                                               *|
|* Revision       Date        Engineer; Reason                   *|
|* --------       ----        ----------------                   *|
|*   0.0         991119        jg;  Created                      *|
\****************************************************************/

#include <malloc.h>
#include <math.h>
#include "fft.h"
#include "ppc_vmx.h"

#define  TWOPI  (double)6.2831853071795864769252868
#define  BITR( log2x, index, bitr_index ) \
   { \
      ulong  _bitr_i, _bitr_x; \
      _bitr_x = (index); \
      bitr_index = 0; \
      for ( _bitr_i = 0; _bitr_i < (log2x); _bitr_i++ ) { \
         bitr_index <<= 1; \
         bitr_index |= (_bitr_x & 1); \
         _bitr_x >>= 1; \
      } \
   }

extern uchar _fft_bitr_1[];
extern uchar _fft_bitr_2[];
extern uchar _fft_bitr_4[];
extern uchar _fft_bitr_8[];
extern uchar _fft_bitr_16[];
extern uchar _fft_bitr_32[];
extern uchar _fft_bitr_64[];
extern uchar _fft_bitr_128[];
extern uchar _fft_bitr_256[];

void  fft_setup( ulong LOG2N, FFT_setup *SETUP )
```

```
{
   char  **mallocp;
   char  *buffer;
   float  *twidp;
   ulong  bitr_i, i, j, log2n_m4, n, nv16;
   double  angle, cos1, cos2, delta, incr, sin1, sin2, twopivn;

   n = 1 << LOG2N;

   buffer = malloc( (n * sizeof(float)) + 20 );
   if ( !buffer ) {
      SETUP->twidp = (float *)0;
      return;
   }

   twidp = (float *)((ulong)(buffer + 20) & ~15);
   mallocp = (char **)(twidp - 1);
   *mallocp = buffer;

   nv16 = n >> 4;
   log2n_m4 = LOG2N - 4;
   twopivn = TWOPI / (double)n;
   delta = (double)0.0;

   for ( i = 0; i < nv16; i++ ) {
      for ( j = 0; j < 4; j++ ) {
         incr = delta;
         angle = twopivn * incr;
         cos1 = cos(angle);
         sin1 = sin(angle);
         incr += delta;
         angle = twopivn * incr;
         cos2 = cos(angle);
         sin2 = sin(angle);

         if ( ( i == 0 ) || ( i == (nv16 - 1) ) ) {
            twidp[(i << 4) + j] = (float)cos1;
            twidp[(i << 4) + j + 4] = (float)sin1;
            twidp[(i << 4) + j + 8] = (float)cos2;
            twidp[(i << 4) + j + 12] = (float)sin2;
         }
         else {
            BITR( log2n_m4, i, bitr_i )
            twidp[(bitr_i << 4) + j] = (float)cos1;
            twidp[(bitr_i << 4) + j + 4] = (float)(sin1 / cos1);
            twidp[(bitr_i << 4) + j + 8] = (float)(cos2 / sin2);
            twidp[(bitr_i << 4) + j + 12] = (float)sin2;
         }
         delta += (double)1.0;
      }
   }

   SETUP->twidp = twidp;
   if ( LOG2N == 4 )
      SETUP->bitrp = _fft_bitr_1;
   else if ( LOG2N == 5 )
```

```
      SETUP->bitrp = _fft_bitr_2;
   else if ( LOG2N == 6 )
      SETUP->bitrp = _fft_bitr_4;
   else if ( LOG2N == 7 )
      SETUP->bitrp = _fft_bitr_8;
   else if ( LOG2N == 8 )
      SETUP->bitrp = _fft_bitr_16;
   else if ( LOG2N == 9 )
      SETUP->bitrp = _fft_bitr_32;
   else if ( LOG2N == 10 )
      SETUP->bitrp = _fft_bitr_64;
   else if ( LOG2N == 11 )
      SETUP->bitrp = _fft_bitr_128;
   else if ( LOG2N == 12 )
      SETUP->bitrp = _fft_bitr_256;
   return;
}

void  fft_free( FFT_setup *SETUP )

{
   char  **mallocp;

   if ( (SETUP->bitrp == _fft_bitr_1) ||
        (SETUP->bitrp == _fft_bitr_2) ||
        (SETUP->bitrp == _fft_bitr_4) ||
        (SETUP->bitrp == _fft_bitr_8) ||
        (SETUP->bitrp == _fft_bitr_16) ||
        (SETUP->bitrp == _fft_bitr_32) ||
        (SETUP->bitrp == _fft_bitr_64) ||
        (SETUP->bitrp == _fft_bitr_128) ||
        (SETUP->bitrp == _fft_bitr_256) ) {
      mallocp = (char **)(SETUP->twidp - 1);
      free ( *mallocp );
   }
   SETUP->twidp = (float *)0;
   SETUP->bitrp = (uchar *)0;
   return;
}
```

```
/****************************************************************\
|*  File Name:      fft_z.c                                     *|
|*  Description:    Forward (or Inverse) Complex In-place 1D FFT *|
|*  Entry/params:   void fft_z ( float *Cr, float *Ci,          *|
|*                               ulong LOG2N, FFT_setup *SETUP ) *|
|*                                                               *|
|*  Formula:                                                     *|
|*                                                               *|
|*    Cr/Ci = 2^LOG2N-point (4 <= LOG2N <= 12) forward in-place *|
|*            complex 1d FFT of the split complex vector stored *|
|*            in Cr and Ci.                                     *|
|*                                                               *|
|*      (Note, an inverse FFT can be performed by swapping      *|
|*       Cr and Ci.)                                            *|
|*                                                               *|
|*  where:                                                       *|
|*                                                               *|
|*    Cr and Ci must be 16-byte aligned and have unit stride    *|
|*    stride between adjacent real (Cr) and imaginary (Ci)      *|
|*    points.                                                    *|
|*                                                               *|
|*    LOG2N is the log (base 2) of the FFT size.                *|
|*      (4 <= LOG2N <= 12)                                      *|
|*                                                               *|
|*    Let:  N = 2 ^ LOG2N                                       *|
|*          LOG2M = LOG2N - 4                                   *|
|*          M = 2 ^ LOG2M                                       *|
|*          A = 2 * PI / N                                      *|
|*          BITR( i, m ) = bit-reversal of unsigned integer i   *|
|*                         over m bits                          *|
|*                                                               *|
|*    SETUP->twidp is a 16-byte aligned pointer to M sets       *|
|*      of 4 x 4 floating point twiddles arranged exactly       *|
|*      as follows:                                             *|
|*                                                               *|
|*      cos(kA), cos((k+1)A), cos((k+2)A), cos((k+3)A),         *|
|*      sin(kA), sin((k+1)A), sin((k+2)A), sin((k+3)A),         *|
|*      cos(2kA), cos(2(k+1)A), cos(2(k+2)A), cos(2(k+3)A),     *|
|*      sin(2kA), sin(2(k+1)A), sin(2(k+2)A), sin(2(k+3)A)      *|
|*                                                               *|
|*        for k = 0                                             *|
|*                                                               *|
|*      cos(kA), cos((k+1)A), cos((k+2)A), cos((k+3)A),         *|
|*      tan(kA), tan((k+1)A), tan((k+2)A), tan((k+3)A),         *|
|*      cot(2kA), cot(2(k+1)A), cot(2(k+2)A), cot(2(k+3)A),     *|
|*      sin(2kA), sin(2(k+1)A), sin(2(k+2)A), sin(2(k+3)A)      *|
|*                                                               *|
|*        for k = 4 * BITR( 1, LOG2M ),                         *|
|*                4 * BITR( 2, LOG2M ),                         *|
|*                ...,                                          *|
|*                4 * BITR( M-2, LOG2M )                        *|
|*                                                               *|
|*      cos(kA), cos((k+1)A), cos((k+2)A), cos((k+3)A),         *|
|*      sin(kA), sin((k+1)A), sin((k+2)A), sin((k+3)A),         *|
|*      cos(2kA), cos(2(k+1)A), cos(2(k+2)A), cos(2(k+3)A),     *|
|*      sin(2kA), sin(2(k+1)A), sin(2(k+2)A), sin(2(k+3)A)      *|
```

```
|*                                                                  *|
|*        for k = 4 * (M - 1)                                      *|
|*                                                                  *|
|*    SETUP->bitrp is a pointer to M unsigned char                 *|
|*      bit-reversed index values (LOG2M bits) arranged            *|
|*      as follows:                                                *|
|*                                                                  *|
|*      section 1:                                                 *|
|*        n1 = bitrp[0] = # of elements in section 1               *|
|*        (The first and second elements are not in the table      *|
|*        as they are known to be 0 and M-1, respectively.)        *|
|*                                                                  *|
|*        0, M-1, bitrp[1], ..., bitrp[n1-2] =                     *|
|*        indices that bit-reverse to themselves                   *|
|*                                                                  *|
|*      section 2:                                                 *|
|*        n2 = bitrp[n1-1] = # of elements in section 2            *|
|*        It's always true that n1 + n2 = M.                       *|
|*        (The first element is not in the table and, if           *|
|*        n2 != 0, is known to be 1.)                              *|
|*                                                                  *|
|*        (1, bitrp[n1]), (bitrp[n1+1], bitrp[n1+2]), ...,         *|
|*        (bitrp[M-3], bitrp[M-2]) = n2/2 pairs of indices that    *|
|*        bit-reverse to each other. bitrp[M-1] = 0.               *|
|*                                                                  *|
|*            Mercury Computer Systems, Inc.                       *|
|*            Copyright (c) 1999  All rights reserved              *|
|*                                                                  *|
|* Revision      Date         Engineer; Reason                     *|
|* --------      ----         ----------------                     *|
|*   0.0        991119        jg;  Created                         *|
\*******************************************************************/

#include "fft.h"
#include "ppc_vmx.h"

/*
 *  _fft_z
 */

void fft_z ( float *Cr, float *Ci, ulong LOG2N, FFT_setup *SETUP )

{
   float  *Cr1, *Ci1, *Cr2, *Ci2, *Cr3, *Ci3;
   float  *Cr4, *Ci4, *Cr5, *Ci5, *Cr6, *Ci6, *Cr7, *Ci7;
   float  *wp0, *wp1, *wp2, *wp3;
   unsigned char  *bitrp;

   ulong  index, index_bump, index1, index2, windex;
   ulong  bflycnt, bflyoff, gcnt, scnt, N;

   VMX_reg  a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i;
   VMX_reg  y0r, y0i, y1r, y1i, y2r, y2i, y3r, y3i;
   VMX_reg  t1r, t1i, t2r, t2i, m2r, m2i, m3r, m3i;
   VMX_reg  p0r, p0i, p1r, p1i, p2r, p2i, p3r, p3i;
   VMX_reg  x1r, x1i, x2r, x2i;
   VMX_reg  cos1, sin1, cos2, sin2, tan1, cot2;
```

```
VMX_reg  a0r_8, a0i_8, a1r_8, a1i_8, a2r_8, a2i_8, a3r_8, a3i_8;
VMX_reg  a4r_8, a4i_8, a5r_8, a5i_8, a6r_8, a6i_8, a7r_8, a7i_8;
VMX_reg  y0r_8, y0i_8, y1r_8, y1i_8, y2r_8, y2i_8, y3r_8, y3i_8;
VMX_reg  y4r_8, y4i_8, y5r_8, y5i_8, y6r_8, y6i_8, y7r_8, y7i_8;
VMX_reg  t1r_8, t1i_8, t2r_8, t2i_8, t3r_8, t3i_8, t4r_8, t4i_8;
VMX_reg  t5r_8, t5i_8, t6r_8, t6i_8, t7r_8, t7i_8, t8r_8, t8i_8;
VMX_reg  d1r_8, d1i_8, d2r_8, d2i_8, m2r_8, m2i_8, m5r_8, m5i_8;
VMX_reg  s1r_8, s1i_8, s2r_8, s2i_8, s3r_8, s3i_8, s4r_8, s4i_8;
VMX_reg  em4r_8, em4i_8, em7r_8, em7i_8, rad2v2;

/*
 *  here if N >= 16
 */
wp0 = SETUP->twidp;
wp1 = wp0 + 4;
wp2 = wp0 + 8;
wp3 = wp0 + 12;
bitrp = SETUP->bitrp;
N = 1 << LOG2N;

if ( LOG2N & 1 ) {

   /* radix-8 first pass */

   windex = 64;
   LVEWX( rad2v2, wp0, windex )      /* cos (PI/4) = sqrt(2)/2 */
   bflyoff = N >> 1;                 /* 4 * N/8 = N/2 byte offset */
   VSPLTW( rad2v2, rad2v2, 0 )       /* replicate 4 times */

   Cr1 = (float *)((char *)Cr + bflyoff);
   Ci1 = (float *)((char *)Ci + bflyoff);
   Cr2 = (float *)((char *)Cr1 + bflyoff);
   Ci2 = (float *)((char *)Ci1 + bflyoff);
   Cr3 = (float *)((char *)Cr2 + bflyoff);
   Ci3 = (float *)((char *)Ci2 + bflyoff);
   Cr4 = (float *)((char *)Cr3 + bflyoff);
   Ci4 = (float *)((char *)Ci3 + bflyoff);
   Cr5 = (float *)((char *)Cr4 + bflyoff);
   Ci5 = (float *)((char *)Ci4 + bflyoff);
   Cr6 = (float *)((char *)Cr5 + bflyoff);
   Ci6 = (float *)((char *)Ci5 + bflyoff);
   Cr7 = (float *)((char *)Cr6 + bflyoff);
   Ci7 = (float *)((char *)Ci6 + bflyoff);

   index = 0;

   bflycnt = bflyoff;
   while ( bflycnt ) {               /* while ( index < bflyoff ) { */
      LVX( a0r_8, Cr, index )
      LVX( a0i_8, Ci, index )
      LVX( a1r_8, Cr1, index )
      LVX( a1i_8, Ci1, index )
      LVX( a2r_8, Cr2, index )
      LVX( a2i_8, Ci2, index )
      LVX( a3r_8, Cr3, index )
      LVX( a3i_8, Ci3, index )
      LVX( a4r_8, Cr4, index )
```

```
LVX( a4i_8, Ci4, index )
LVX( a5r_8, Cr5, index )
LVX( a5i_8, Ci5, index )
LVX( a6r_8, Cr6, index )
LVX( a6i_8, Ci6, index )
LVX( a7r_8, Cr7, index )
LVX( a7i_8, Ci7, index )

VADDFP( t1r_8, a0r_8, a4r_8 )
VSUBFP( d1r_8, a0r_8, a4r_8 )
VADDFP( t1i_8, a0i_8, a4i_8 )
VSUBFP( d1i_8, a0i_8, a4i_8 )

VADDFP( t3r_8, a1r_8, a5r_8 )
VSUBFP( t4r_8, a5r_8, a1r_8 )
VADDFP( t3i_8, a1i_8, a5i_8 )
VSUBFP( t4i_8, a1i_8, a5i_8 )

VADDFP( t2r_8, a2r_8, a6r_8 )
VSUBFP( d2r_8, a6r_8, a2r_8 )
VADDFP( t2i_8, a2i_8, a6i_8 )
VSUBFP( d2i_8, a2i_8, a6i_8 )

VADDFP( t5r_8, a3r_8, a7r_8 )
VSUBFP( t6r_8, a7r_8, a3r_8 )
VADDFP( t5i_8, a3i_8, a7i_8 )
VSUBFP( t6i_8, a3i_8, a7i_8 )

VADDFP( t7r_8, t1r_8, t2r_8 )
VSUBFP( m2r_8, t1r_8, t2r_8 )
VADDFP( t7i_8, t1i_8, t2i_8 )
VSUBFP( m2i_8, t1i_8, t2i_8 )

VADDFP( t8r_8, t5r_8, t3r_8 )
VADDFP( t8i_8, t3i_8, t5i_8 )
VSUBFP( m5r_8, t3i_8, t5i_8 )
VSUBFP( m5i_8, t5r_8, t3r_8 )

VADDFP( y0r_8, t7r_8, t8r_8 )
VADDFP( y0i_8, t7i_8, t8i_8 )
VADDFP( y2r_8, m2r_8, m5r_8 )
VADDFP( y2i_8, m2i_8, m5i_8 )

VSUBFP( y4r_8, t7r_8, t8r_8 )
VSUBFP( y4i_8, t7i_8, t8i_8 )
VSUBFP( y6r_8, m2r_8, m5r_8 )
VSUBFP( y6i_8, m2i_8, m5i_8 )

VSUBFP( em4r_8, t6r_8, t4r_8 )
VSUBFP( em4i_8, t4i_8, t6i_8 )
VADDFP( em7r_8, t4i_8, t6i_8 )
VADDFP( em7i_8, t6r_8, t4r_8 )

VMADDFP( s1r_8, rad2v2, em4r_8, d1r_8 )
VMADDFP( s1i_8, rad2v2, em4i_8, d1i_8 )
VNMSUBFP( s2r_8, rad2v2, em4r_8, d1r_8 )
VNMSUBFP( s2i_8, rad2v2, em4i_8, d1i_8 )
```

```
        VMADDFP( s3r_8, rad2v2, em7r_8, d2i_8 )
        VMADDFP( s3i_8, rad2v2, em7i_8, d2r_8 )
        VNMSUBFP( s4r_8, rad2v2, em7r_8, d2i_8 )
        VNMSUBFP( s4i_8, rad2v2, em7i_8, d2r_8 )

        VADDFP( y1r_8, s1r_8, s3r_8 )
        VADDFP( y1i_8, s1i_8, s3i_8 )
        VSUBFP( y3r_8, s2r_8, s4r_8 )
        VSUBFP( y3i_8, s2i_8, s4i_8 )

        VADDFP( y5r_8, s2r_8, s4r_8 )
        VADDFP( y5i_8, s2i_8, s4i_8 )
        VSUBFP( y7r_8, s1r_8, s3r_8 )
        VSUBFP( y7i_8, s1i_8, s3i_8 )

        STVX( y0r_8, Cr, index )       /* bit-reverse output */
        STVX( y0i_8, Ci, index )
        STVX( y2r_8, Cr2, index )
        STVX( y2i_8, Ci2, index )
        STVX( y4r_8, Cr1, index )
        STVX( y4i_8, Ci1, index )
        STVX( y6r_8, Cr3, index )
        STVX( y6i_8, Ci3, index )
        STVX( y1r_8, Cr4, index )
        STVX( y1i_8, Ci4, index )
        STVX( y3r_8, Cr6, index )
        STVX( y3i_8, Ci6, index )
        STVX( y5r_8, Cr5, index )
        STVX( y5i_8, Ci5, index )
        STVX( y7r_8, Cr7, index )
        STVX( y7i_8, Ci7, index )

        index += 16;
        bflycnt -= 16;
     }
  }                                    /* end radix-8 first pass */

  else {                               /* radix-4 first pass */

     bflyoff = N;                      /* 4 * N/4 = N byte offset */

     Cr1 = (float *)((char *)Cr + bflyoff);
     Ci1 = (float *)((char *)Ci + bflyoff);
     Cr2 = (float *)((char *)Cr1 + bflyoff);
     Ci2 = (float *)((char *)Ci1 + bflyoff);
     Cr3 = (float *)((char *)Cr2 + bflyoff);
     Ci3 = (float *)((char *)Ci2 + bflyoff);

     index = 0;

     bflycnt = bflyoff;
     while ( bflycnt ) {               /* while ( index < bflyoff ) { */
        LVX( a0r, Cr, index )
        LVX( a0i, Ci, index )
        LVX( a1r, Cr1, index )
        LVX( a1i, Ci1, index )
```

```
            LVX( a2r, Cr2, index )
            LVX( a2i, Ci2, index )
            LVX( a3r, Cr3, index )
            LVX( a3i, Ci3, index )

            VADDFP( t1r, a0r, a2r )
            VADDFP( t1i, a0i, a2i )
            VSUBFP( m2r, a0r, a2r )
            VSUBFP( m2i, a0i, a2i )

            VADDFP( t2r, a3r, a1r )
            VADDFP( t2i, a1i, a3i )
            VSUBFP( m3r, a1i, a3i )
            VSUBFP( m3i, a3r, a1r )

            VADDFP( y0r, t1r, t2r )
            VADDFP( y0i, t1i, t2i )
            VADDFP( y1r, m2r, m3r )
            VADDFP( y1i, m2i, m3i )

            VSUBFP( y2r, t1r, t2r )
            VSUBFP( y2i, t1i, t2i )
            VSUBFP( y3r, m2r, m3r )
            VSUBFP( y3i, m2i, m3i )

            STVX( y0r, Cr, index )          /* bit-reverse output */
            STVX( y0i, Ci, index )
            STVX( y1r, Cr2, index )
            STVX( y1i, Ci2, index )
            STVX( y2r, Cr1, index )
            STVX( y2i, Ci1, index )
            STVX( y3r, Cr3, index )
            STVX( y3i, Ci3, index )

            index += 16;
            bflycnt -= 16;
        }
    }                                       /* end radix-4 first pass */

    while ( bflyoff > 64 ) {                /* middle stages */

        index_bump = bflyoff;
        bflyoff >>= 2;                      /* decimate by 4 */
        index_bump -= bflyoff;              /* 3 * bflyoff */

        Cr1 = (float *)((char *)Cr + bflyoff);   /* adjust pointers */
        Ci1 = (float *)((char *)Ci + bflyoff);
        Cr2 = (float *)((char *)Cr1 + bflyoff);
        Ci2 = (float *)((char *)Ci1 + bflyoff);
        Cr3 = (float *)((char *)Cr2 + bflyoff);
        Ci3 = (float *)((char *)Ci2 + bflyoff);

        index = 0;

        bflycnt = bflyoff;
        while ( bflycnt ) {                 /* first (weightless) group */
            LVX( a0r, Cr, index )
```

```
        LVX( a0i, Ci, index )
        LVX( a1r, Cr1, index )
        LVX( a1i, Ci1, index )
        LVX( a2r, Cr2, index )
        LVX( a2i, Ci2, index )
        LVX( a3r, Cr3, index )
        LVX( a3i, Ci3, index )

        VADDFP( t1r, a0r, a2r )
        VADDFP( t1i, a0i, a2i )
        VSUBFP( m2r, a0r, a2r )
        VSUBFP( m2i, a0i, a2i )

        VADDFP( t2r, a3r, a1r )
        VADDFP( t2i, a1i, a3i )
        VSUBFP( m3r, a1i, a3i )
        VSUBFP( m3i, a3r, a1r )

        VADDFP( y0r, t1r, t2r )
        VADDFP( y0i, t1i, t2i )
        VADDFP( y1r, m2r, m3r )
        VADDFP( y1i, m2i, m3i )

        VSUBFP( y2r, t1r, t2r )
        VSUBFP( y2i, t1i, t2i )
        VSUBFP( y3r, m2r, m3r )
        VSUBFP( y3i, m2i, m3i )

        STVX( y0r, Cr, index )         /* bit-reverse output */
        STVX( y0i, Ci, index )
        STVX( y1r, Cr2, index )
        STVX( y1i, Ci2, index )
        STVX( y2r, Cr1, index )
        STVX( y2i, Ci1, index )
        STVX( y3r, Cr3, index )
        STVX( y3i, Ci3, index )

        index += 16;
        bflycnt -= 16;
    }                                  /* end of first (weightless) group */

    windex = 64;

    gcnt = N - bflyoff;
    while ( gcnt ) {                   /* loop for remaining groups */

        /*
         *  load weights for group
         */
        LVEWX( cos1, wp0, windex )
        LVEWX( tan1, wp1, windex )
        LVEWX( cot2, wp2, windex )
        LVEWX( sin2, wp3, windex )
        VSPLTW( cos1, cos1, 0 )        /* replicate 4 times */
        VSPLTW( tan1, tan1, 0 )
        VSPLTW( cot2, cot2, 0 )
        VSPLTW( sin2, sin2, 0 )
```

```
index += index_bump;

bflycnt = bflyoff;
while ( bflycnt ) {

   LVX( a0r, Cr, index )
   LVX( a0i, Ci, index )
   LVX( a1r, Cr1, index )
   LVX( a1i, Ci1, index )
   LVX( a2r, Cr2, index )
   LVX( a2i, Ci2, index )
   LVX( a3r, Cr3, index )
   LVX( a3i, Ci3, index )

   VMADDFP( x1r, cot2, a2r, a2i )
   VNMSUBFP( x1i, cot2, a2i, a2r )
   VMADDFP( x2r, cot2, a3r, a3i )
   VNMSUBFP( x2i, cot2, a3i, a3r )

   VMADDFP( t1r, sin2, x1r, a0r )
   VNMSUBFP( t1i, sin2, x1i, a0i )
   VMADDFP( t2r, sin2, x2r, a1r )
   VNMSUBFP( t2i, sin2, x2i, a1i )

   VNMSUBFP( m2r, sin2, x1r, a0r )
   VMADDFP( m2i, sin2, x1i, a0i )
   VNMSUBFP( m3r, sin2, x2r, a1r )
   VMADDFP( m3i, sin2, x2i, a1i )

   VMADDFP( x1r, tan1, t2i, t2r )
   VNMSUBFP( x1i, tan1, t2r, t2i )
   VNMSUBFP( x2r, tan1, m3r, m3i )
   VMADDFP( x2i, tan1, m3i, m3r )

   VMADDFP( y0r, cos1, x1r, t1r )
   VMADDFP( y0i, cos1, x1i, t1i )
   VMADDFP( y1r, cos1, x2r, m2r )
   VNMSUBFP( y1i, cos1, x2i, m2i )

   VNMSUBFP( y2r, cos1, x1r, t1r )
   VNMSUBFP( y2i, cos1, x1i, t1i )
   VNMSUBFP( y3r, cos1, x2r, m2r )
   VMADDFP( y3i, cos1, x2i, m2i )

   STVX( y0r, Cr, index )      /* bit-reverse output */
   STVX( y0i, Ci, index )
   STVX( y1r, Cr2, index )
   STVX( y1i, Ci2, index )
   STVX( y2r, Cr1, index )
   STVX( y2i, Ci1, index )
   STVX( y3r, Cr3, index )
   STVX( y3i, Ci3, index )

   index += 16;
   bflycnt -= 16;
}                              /* end of butterfly loop */
```

```
        windex += 64;                   /* bump weight index */
        gcnt -= bflyoff;
      }                                 /* end of group loop */
    }                                   /* end of stage loop */

    if ( bflyoff == 64 ) {              /* penultimate stage */

      Cr1 = (float *)((char *)Cr + 16);     /* adjust pointers */
      Ci1 = (float *)((char *)Ci + 16);
      Cr2 = (float *)((char *)Cr1 + 16);
      Ci2 = (float *)((char *)Ci1 + 16);
      Cr3 = (float *)((char *)Cr2 + 16);
      Ci3 = (float *)((char *)Ci2 + 16);

      index = 0;                        /* same as windex */

      /*
       *  first group (4 butterflies) is weightless
       */
      LVX( a0r, Cr, index )
      LVX( a0i, Ci, index )
      LVX( a1r, Cr1, index )
      LVX( a1i, Ci1, index )
      LVX( a2r, Cr2, index )
      LVX( a2i, Ci2, index )
      LVX( a3r, Cr3, index )
      LVX( a3i, Ci3, index )

      VADDFP( t1r, a0r, a2r )
      VADDFP( t1i, a0i, a2i )
      VSUBFP( m2r, a0r, a2r )
      VSUBFP( m2i, a0i, a2i )

      VADDFP( t2r, a3r, a1r )
      VADDFP( t2i, a1i, a3i )
      VSUBFP( m3r, a1i, a3i )
      VSUBFP( m3i, a3r, a1r )

      VADDFP( y0r, t1r, t2r )
      VADDFP( y0i, t1i, t2i )
      VADDFP( y1r, m2r, m3r )
      VADDFP( y1i, m2i, m3i )

      VSUBFP( y2r, t1r, t2r )
      VSUBFP( y2i, t1i, t2i )
      VSUBFP( y3r, m2r, m3r )
      VSUBFP( y3i, m2i, m3i )

      STVX( y0r, Cr, index )            /* bit-reverse output */
      STVX( y0i, Ci, index )
      STVX( y1r, Cr2, index )
      STVX( y1i, Ci2, index )
      STVX( y2r, Cr1, index )
      STVX( y2i, Ci1, index )
      STVX( y3r, Cr3, index )
      STVX( y3i, Ci3, index )
```

```
/*
 * loop for remaining butterflies except the very last
 */
bflycnt = N - 32;
while ( bflycnt ) {

   index += 64;

   /*
    *  load weights for group
    */
   LVEWX( cos1, wp0, index )
   LVEWX( tan1, wp1, index )
   LVEWX( cot2, wp2, index )
   LVEWX( sin2, wp3, index )
   VSPLTW( cos1, cos1, 0 )        /* replicate 4 times */
   VSPLTW( tan1, tan1, 0 )
   VSPLTW( cot2, cot2, 0 )
   VSPLTW( sin2, sin2, 0 )

   LVX( a0r, Cr, index )
   LVX( a0i, Ci, index )
   LVX( a1r, Cr1, index )
   LVX( a1i, Ci1, index )
   LVX( a2r, Cr2, index )
   LVX( a2i, Ci2, index )
   LVX( a3r, Cr3, index )
   LVX( a3i, Ci3, index )

   VMADDFP( x1r, cot2, a2r, a2i )
   VNMSUBFP( x1i, cot2, a2i, a2r )
   VMADDFP( x2r, cot2, a3r, a3i )
   VNMSUBFP( x2i, cot2, a3i, a3r )

   VMADDFP( t1r, sin2, x1r, a0r )
   VNMSUBFP( t1i, sin2, x1i, a0i )
   VMADDFP( t2r, sin2, x2r, a1r )
   VNMSUBFP( t2i, sin2, x2i, a1i )

   VNMSUBFP( m2r, sin2, x1r, a0r )
   VMADDFP( m2i, sin2, x1i, a0i )
   VNMSUBFP( m3r, sin2, x2r, a1r )
   VMADDFP( m3i, sin2, x2i, a1i )

   VMADDFP( x1r, tan1, t2i, t2r )
   VNMSUBFP( x1i, tan1, t2r, t2i )
   VNMSUBFP( x2r, tan1, m3r, m3i )
   VMADDFP( x2i, tan1, m3i, m3r )

   VMADDFP( y0r, cos1, x1r, t1r )
   VMADDFP( y0i, cos1, x1i, t1i )
   VMADDFP( y1r, cos1, x2r, m2r )
   VNMSUBFP( y1i, cos1, x2i, m2i )

   VNMSUBFP( y2r, cos1, x1r, t1r )
   VNMSUBFP( y2i, cos1, x1i, t1i )
   VNMSUBFP( y3r, cos1, x2r, m2r )
```

```
    VMADDFP( y3i, cos1, x2i, m2i )

    STVX( y0r, Cr, index )          /* bit-reverse output */
    STVX( y0i, Ci, index )
    STVX( y1r, Cr2, index )
    STVX( y1i, Ci2, index )
    STVX( y2r, Cr1, index )
    STVX( y2i, Ci1, index )
    STVX( y3r, Cr3, index )
    STVX( y3i, Ci3, index )

    bflycnt -= 16;
}                                   /* end of butterfly loop */

/*
 *  very last butterfly uses cosine/sine weights for accuracy
 */
index += 64;

LVEWX( cos1, wp0, index )
LVEWX( sin1, wp1, index )
LVEWX( cos2, wp2, index )
LVEWX( sin2, wp3, index )
VSPLTW( cos1, cos1, 0 )             /* replicate 4 times */
VSPLTW( sin1, sin1, 0 )
VSPLTW( cos2, cos2, 0 )
VSPLTW( sin2, sin2, 0 )

LVX( a1r, Cr1, index )
LVX( a1i, Ci1, index )
LVX( a2r, Cr2, index )
LVX( a2i, Ci2, index )
LVX( a3r, Cr3, index )
LVX( a3i, Ci3, index )
LVX( a0r, Cr, index )
LVX( a0i, Ci, index )

VMADDFP( t1r, cos2, a2r, a0r )
VMADDFP( t1i, cos2, a2i, a0i )
VNMSUBFP( m2r, cos2, a2r, a0r )
VNMSUBFP( m2i, cos2, a2i, a0i )

VMADDFP( t1r, sin2, a2i, t1r )
VNMSUBFP( t1i, sin2, a2r, t1i )
VNMSUBFP( m2r, sin2, a2i, m2r )
VMADDFP( m2i, sin2, a2r, m2i )

VMADDFP( t2r, cos2, a3r, a1r )
VMADDFP( t2i, cos2, a3i, a1i )
VNMSUBFP( m3r, cos2, a3r, a1r )
VNMSUBFP( m3i, cos2, a3i, a1i )

VMADDFP( t2r, sin2, a3i, t2r )
VNMSUBFP( t2i, sin2, a3r, t2i )
VNMSUBFP( m3r, sin2, a3i, m3r )
VMADDFP( m3i, sin2, a3r, m3i )
```

```
    VMADDFP( y0r, cos1, t2r, t1r )
    VMADDFP( y0i, cos1, t2i, t1i )
    VNMSUBFP( y2r, cos1, t2r, t1r )
    VNMSUBFP( y2i, cos1, t2i, t1i )

    VMADDFP( y0r, sin1, t2i, y0r )
    VNMSUBFP( y0i, sin1, t2r, y0i )
    VNMSUBFP( y2r, sin1, t2i, y2r )
    VMADDFP( y2i, sin1, t2r, y2i )

    VNMSUBFP( y1r, sin1, m3r, m2r )
    VNMSUBFP( y1i, sin1, m3i, m2i )
    VMADDFP( y3r, sin1, m3r, m2r )
    VMADDFP( y3i, sin1, m3i, m2i )

    VMADDFP( y1r, cos1, m3i, y1r )
    VNMSUBFP( y1i, cos1, m3r, y1i )
    VNMSUBFP( y3r, cos1, m3i, y3r )
    VMADDFP( y3i, cos1, m3r, y3i )

    STVX( y0r, Cr, index )            /* bit-reverse output */
    STVX( y0i, Ci, index )
    STVX( y1r, Cr2, index )
    STVX( y1i, Ci2, index )
    STVX( y2r, Cr1, index )
    STVX( y2i, Ci1, index )
    STVX( y3r, Cr3, index )
    STVX( y3i, Ci3, index )
}                                     /* end penultimate pass */

/*
 *  final pass
 */
Cr1 = (float *)((char *)Cr + N);      /* adjust pointers */
Ci1 = (float *)((char *)Ci + N);
Cr2 = (float *)((char *)Cr1 + N);
Ci2 = (float *)((char *)Ci1 + N);
Cr3 = (float *)((char *)Cr2 + N);
Ci3 = (float *)((char *)Ci2 + N);

bflycnt = (ulong)*bitrp;
windex = 0;
index = 0;

scnt = (bflycnt == 1) ? 1 : 2;
bflycnt -= scnt;

/*
 *  loop for in-place butterflies using cosine/sine weights (at most 2)
 */
while ( scnt ) {

   LVX( a0r, Cr, index )
   LVX( a0i, Ci, index )
   LVX( a1r, Cr1, index )
   LVX( a1i, Ci1, index )
   LVX( a2r, Cr2, index )
```

```
LVX( a2i, Ci2, index )
LVX( a3r, Cr3, index )
LVX( a3i, Ci3, index )

LVX( cos1, wp0, windex )
LVX( sin1, wp1, windex )
LVX( cos2, wp2, windex )
LVX( sin2, wp3, windex )

/*
 *  perform two (real and imaginary) 4 x 4 permutes
 *  but swapping the resulting 2 middle columns
 */
VMRGHW( p0r, a0r, a1r )
VMRGHW( p0i, a0i, a1i )
VMRGHW( p1r, a2r, a3r )
VMRGHW( p1i, a2i, a3i )

VMRGLW( p2r, a0r, a1r )
VMRGLW( p2i, a0i, a1i )
VMRGLW( p3r, a2r, a3r )
VMRGLW( p3i, a2i, a3i )

VMRGHW( a0r, p0r, p1r )
VMRGHW( a0i, p0i, p1i )
VMRGLW( a1r, p0r, p1r )
VMRGLW( a1i, p0i, p1i )

VMRGHW( a2r, p2r, p3r )
VMRGHW( a2i, p2i, p3i )
VMRGLW( a3r, p2r, p3r )
VMRGLW( a3i, p2i, p3i )

VMADDFP( t1r, cos2, a2r, a0r )
VMADDFP( t1i, cos2, a2i, a0i )
VNMSUBFP( m2r, cos2, a2r, a0r )
VNMSUBFP( m2i, cos2, a2i, a0i )

VMADDFP( t1r, sin2, a2i, t1r )
VNMSUBFP( t1i, sin2, a2r, t1i )
VNMSUBFP( m2r, sin2, a2i, m2r )
VMADDFP( m2i, sin2, a2r, m2i )

VMADDFP( t2r, cos2, a3r, a1r )
VMADDFP( t2i, cos2, a3i, a1i )
VNMSUBFP( m3r, cos2, a3r, a1r )
VNMSUBFP( m3i, cos2, a3i, a1i )

VMADDFP( t2r, sin2, a3i, t2r )
VNMSUBFP( t2i, sin2, a3r, t2i )
VNMSUBFP( m3r, sin2, a3i, m3r )
VMADDFP( m3i, sin2, a3r, m3i )

VMADDFP( y0r, cos1, t2r, t1r )
VMADDFP( y0i, cos1, t2i, t1i )
VNMSUBFP( y2r, cos1, t2r, t1r )
VNMSUBFP( y2i, cos1, t2i, t1i )
```

```
        VMADDFP( y0r, sin1, t2i, y0r )
        VNMSUBFP( y0i, sin1, t2r, y0i )
        VNMSUBFP( y2r, sin1, t2i, y2r )
        VMADDFP( y2i, sin1, t2r, y2i )

        VNMSUBFP( y1r, sin1, m3r, m2r )
        VNMSUBFP( y1i, sin1, m3i, m2i )
        VMADDFP( y3r, sin1, m3r, m2r )
        VMADDFP( y3i, sin1, m3i, m2i )

        VMADDFP( y1r, cos1, m3i, y1r )
        VNMSUBFP( y1i, cos1, m3r, y1i )
        VNMSUBFP( y3r, cos1, m3i, y3r )
        VMADDFP( y3i, cos1, m3r, y3i )

        STVX( y0r, Cr, index )          /* no bit-reversal ! */
        STVX( y0i, Ci, index )
        STVX( y1r, Cr1, index )
        STVX( y1i, Ci1, index )
        STVX( y2r, Cr2, index )
        STVX( y2i, Ci2, index )
        STVX( y3r, Cr3, index )
        STVX( y3i, Ci3, index )

        index = N - 16;
        windex = index << 2;
        scnt -= 1;
    }                                   /* end butterfly loop */

    index = (ulong)*++bitrp;
    windex = index << 6;
    index <<= 4;

    /*
     *  loop for remaining in-place butterflies (uses tan, cot weights)
     */
    while ( bflycnt ) {

        LVX( a0r, Cr, index )
        LVX( a0i, Ci, index )
        LVX( a1r, Cr1, index )
        LVX( a1i, Ci1, index )
        LVX( a2r, Cr2, index )
        LVX( a2i, Ci2, index )
        LVX( a3r, Cr3, index )
        LVX( a3i, Ci3, index )

        LVX( cos1, wp0, windex )
        LVX( tan1, wp1, windex )
        LVX( cot2, wp2, windex )
        LVX( sin2, wp3, windex )

        /*
         *  perform two (real and imaginary) 4 x 4 permutes
         *  but swapping the resulting 2 middle columns
         */
```

```
VMRGHW( p0r, a0r, a1r )
VMRGHW( p0i, a0i, a1i )
VMRGHW( p1r, a2r, a3r )
VMRGHW( p1i, a2i, a3i )

VMRGLW( p2r, a0r, a1r )
VMRGLW( p2i, a0i, a1i )
VMRGLW( p3r, a2r, a3r )
VMRGLW( p3i, a2i, a3i )

VMRGHW( a0r, p0r, p1r )
VMRGHW( a0i, p0i, p1i )
VMRGLW( a1r, p0r, p1r )
VMRGLW( a1i, p0i, p1i )

VMRGHW( a2r, p2r, p3r )
VMRGHW( a2i, p2i, p3i )
VMRGLW( a3r, p2r, p3r )
VMRGLW( a3i, p2i, p3i )

VMADDFP( x1r, cot2, a2r, a2i )
VNMSUBFP( x1i, cot2, a2i, a2r )
VMADDFP( x2r, cot2, a3r, a3i )
VNMSUBFP( x2i, cot2, a3i, a3r )

VMADDFP( t1r, sin2, x1r, a0r )
VNMSUBFP( t1i, sin2, x1i, a0i )
VMADDFP( t2r, sin2, x2r, a1r )
VNMSUBFP( t2i, sin2, x2i, a1i )

VNMSUBFP( m2r, sin2, x1r, a0r )
VMADDFP( m2i, sin2, x1i, a0i )
VNMSUBFP( m3r, sin2, x2r, a1r )
VMADDFP( m3i, sin2, x2i, a1i )

VMADDFP( x1r, tan1, t2i, t2r )
VNMSUBFP( x1i, tan1, t2r, t2i )
VNMSUBFP( x2r, tan1, m3r, m3i )
VMADDFP( x2i, tan1, m3i, m3r )

VMADDFP( y0r, cos1, x1r, t1r )
VMADDFP( y0i, cos1, x1i, t1i )
VMADDFP( y1r, cos1, x2r, m2r )
VNMSUBFP( y1i, cos1, x2i, m2i )

VNMSUBFP( y2r, cos1, x1r, t1r )
VNMSUBFP( y2i, cos1, x1i, t1i )
VNMSUBFP( y3r, cos1, x2r, m2r )
VMADDFP( y3i, cos1, x2i, m2i )

STVX( y0r, Cr, index )          /* no bit-reversal ! */
STVX( y0i, Ci, index )
STVX( y1r, Cr1, index )
STVX( y1i, Ci1, index )
STVX( y2r, Cr2, index )
STVX( y2i, Ci2, index )
STVX( y3r, Cr3, index )
```

```
      STVX( y3i, Ci3, index )

      index = (ulong)*++bitrp;
      bflycnt -= 1;
      windex = index << 6;
      index <<= 4;
   }                                    /* end butterfly loop */

   /*
    *  loop for out-of-place butterflies
    */
   bflycnt = index >> 4;                /* count of bit-reverse indices */
   windex = 64;
   index1 = 16;
   while ( bflycnt ) {

      LVX( cos1, wp0, windex )
      LVX( tan1, wp1, windex )
      LVX( cot2, wp2, windex )
      LVX( sin2, wp3, windex )

      LVX( a0r, Cr, index1 )
      LVX( a0i, Ci, index1 )
      LVX( a1r, Cr1, index1 )
      LVX( a1i, Ci1, index1 )
      LVX( a2r, Cr2, index1 )
      LVX( a2i, Ci2, index1 )
      LVX( a3r, Cr3, index1 )
      LVX( a3i, Ci3, index1 )

      /*
       *  perform two (real and imaginary) 4 x 4 permutes
       *  but swapping the resulting 2 middle columns
       */
      VMRGHW( p0r, a0r, a1r )
      VMRGHW( p0i, a0i, a1i )
      VMRGHW( p1r, a2r, a3r )
      VMRGHW( p1i, a2i, a3i )

      VMRGLW( p2r, a0r, a1r )
      VMRGLW( p2i, a0i, a1i )
      VMRGLW( p3r, a2r, a3r )
      VMRGLW( p3i, a2i, a3i )

      VMRGHW( a0r, p0r, p1r )
      VMRGHW( a0i, p0i, p1i )
      VMRGLW( a1r, p0r, p1r )
      VMRGLW( a1i, p0i, p1i )

      VMRGHW( a2r, p2r, p3r )
      VMRGHW( a2i, p2i, p3i )
      VMRGLW( a3r, p2r, p3r )
      VMRGLW( a3i, p2i, p3i )

      VMADDFP( x1r, cot2, a2r, a2i )
      VNMSUBFP( x1i, cot2, a2i, a2r )
      VMADDFP( x2r, cot2, a3r, a3i )
```

```
VNMSUBFP( x2i, cot2, a3i, a3r )

VMADDFP( t1r, sin2, x1r, a0r )
VNMSUBFP( t1i, sin2, x1i, a0i )
VMADDFP( t2r, sin2, x2r, a1r )
VNMSUBFP( t2i, sin2, x2i, a1i )

VNMSUBFP( m2r, sin2, x1r, a0r )
VMADDFP( m2i, sin2, x1i, a0i )
VNMSUBFP( m3r, sin2, x2r, a1r )
VMADDFP( m3i, sin2, x2i, a1i )

VMADDFP( x1r, tan1, t2i, t2r )
VNMSUBFP( x1i, tan1, t2r, t2i )
VNMSUBFP( x2r, tan1, m3r, m3i )
VMADDFP( x2i, tan1, m3i, m3r )

VMADDFP( y0r, cos1, x1r, t1r )
VMADDFP( y0i, cos1, x1i, t1i )
VMADDFP( y1r, cos1, x2r, m2r )
VNMSUBFP( y1i, cos1, x2i, m2i )

VNMSUBFP( y2r, cos1, x1r, t1r )
VNMSUBFP( y2i, cos1, x1i, t1i )
VNMSUBFP( y3r, cos1, x2r, m2r )
VMADDFP( y3i, cos1, x2i, m2i )

index2 = (ulong)*++bitrp;
windex = index2 << 6;
index2 <<= 4;

LVX( cos1, wp0, windex )
LVX( tan1, wp1, windex )
LVX( cot2, wp2, windex )
LVX( sin2, wp3, windex )

LVX( a0r, Cr, index2 )
LVX( a0i, Ci, index2 )
LVX( a1r, Cr1, index2 )
LVX( a1i, Ci1, index2 )
LVX( a2r, Cr2, index2 )
LVX( a2i, Ci2, index2 )
LVX( a3r, Cr3, index2 )
LVX( a3i, Ci3, index2 )

STVX( y0r, Cr, index2 )          /* no bit-reversal ! */
STVX( y0i, Ci, index2 )
STVX( y1r, Cr1, index2 )
STVX( y1i, Ci1, index2 )
STVX( y2r, Cr2, index2 )
STVX( y2i, Ci2, index2 )
STVX( y3r, Cr3, index2 )
STVX( y3i, Ci3, index2 )

/*
 *  perform two (real and imaginary) 4 x 4 permutes
 *  but swapping the resulting 2 middle columns
```

```
*/
VMRGHW( p0r, a0r, a1r )
VMRGHW( p0i, a0i, a1i )
VMRGHW( p1r, a2r, a3r )
VMRGHW( p1i, a2i, a3i )

VMRGLW( p2r, a0r, a1r )
VMRGLW( p2i, a0i, a1i )
VMRGLW( p3r, a2r, a3r )
VMRGLW( p3i, a2i, a3i )

VMRGHW( a0r, p0r, p1r )
VMRGHW( a0i, p0i, p1i )
VMRGLW( a1r, p0r, p1r )
VMRGLW( a1i, p0i, p1i )

VMRGHW( a2r, p2r, p3r )
VMRGHW( a2i, p2i, p3i )
VMRGLW( a3r, p2r, p3r )
VMRGLW( a3i, p2i, p3i )

VMADDFP( x1r, cot2, a2r, a2i )
VNMSUBFP( x1i, cot2, a2i, a2r )
VMADDFP( x2r, cot2, a3r, a3i )
VNMSUBFP( x2i, cot2, a3i, a3r )

VMADDFP( t1r, sin2, x1r, a0r )
VNMSUBFP( t1i, sin2, x1i, a0i )
VMADDFP( t2r, sin2, x2r, a1r )
VNMSUBFP( t2i, sin2, x2i, a1i )

VNMSUBFP( m2r, sin2, x1r, a0r )
VMADDFP( m2i, sin2, x1i, a0i )
VNMSUBFP( m3r, sin2, x2r, a1r )
VMADDFP( m3i, sin2, x2i, a1i )

VMADDFP( x1r, tan1, t2i, t2r )
VNMSUBFP( x1i, tan1, t2r, t2i )
VNMSUBFP( x2r, tan1, m3r, m3i )
VMADDFP( x2i, tan1, m3i, m3r )

VMADDFP( y0r, cos1, x1r, t1r )
VMADDFP( y0i, cos1, x1i, t1i )
VMADDFP( y1r, cos1, x2r, m2r )
VNMSUBFP( y1i, cos1, x2i, m2i )

VNMSUBFP( y2r, cos1, x1r, t1r )
VNMSUBFP( y2i, cos1, x1i, t1i )
VNMSUBFP( y3r, cos1, x2r, m2r )
VMADDFP( y3i, cos1, x2i, m2i )

STVX( y0r, Cr, index1 )          /* no bit-reversal ! */
STVX( y0i, Ci, index1 )
STVX( y1r, Cr1, index1 )
STVX( y1i, Ci1, index1 )
STVX( y2r, Cr2, index1 )
STVX( y2i, Ci2, index1 )
```

```
        STVX( y3r, Cr3, index1 )
        STVX( y3i, Ci3, index1 )

        index1 = (ulong)*++bitrp;
        windex = index1 << 6;
        index1 <<= 4;

        bflycnt -= 2;
    }                                           /* end butterfly loop */
}
```

```
/*****************************************************************\
|*  File Name:      ppc_vmx.c                                     *|
|*  Description:    Contains C functions that emulate PPC vmx     *|
|*                  (altivec) instructions                        *|
|*                                                                *|
|*             Mercury Computer Systems, Inc.                     *|
|*             Copyright (c) 1999  All rights reserved            *|
|*                                                                *|
|* Revision      Date        Engineer; Reason                     *|
|* --------      ----        ----------------                     *|
|*   0.0        991119        jg;  Created                        *|
\*****************************************************************/

#include "ppc_vmx.h"

long  CR[ 8 ];                          /* condition register */

void _lvewx( VMX_reg *vT, ulong rA, ulong rB )
{
   ulong  *addr;
   ulong  i;
   addr = (ulong *)((rA) + (rB));
   i = ((ulong)addr & 0xc) >> 2;
   (vT)->ul[i] = *addr;
}

void _lvx( VMX_reg *vT, ulong rA, ulong rB )
{
   ulong  *addr;
   ulong  i;
   addr = (ulong *)(((rA) + (rB)) & ~15);
   for ( i = 0; i < 4; i++ )
      (vT)->ul[i] = addr[i];
}

void _stvewx( VMX_reg *vS, ulong rA, ulong rB )
{
   ulong  *addr;
   ulong  i;
   addr = (ulong *)((rA) + (rB));
   i = ((ulong)addr & 0xc) >> 2;
   *addr = (vS)->ul[i];
}

void _stvx( VMX_reg *vS, ulong rA, ulong rB )
{
   ulong  *addr;
   ulong  i;
   addr = (ulong *)(((rA) + (rB)) & ~15);
   for ( i = 0; i < 4; i++ )
      addr[i] = (vS)->ul[i];
}

void _vaddfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB )
{
   ulong  i;
```

```
   for ( i = 0; i < 4; i++ )
      (vT)->f[i] = (vA)->f[i] + (vB)->f[i];
}

void _vmaddfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vC, VMX_reg *vB )
{
   ulong  i;
   for ( i = 0; i < 4; i++ )
      (vT)->f[i] = ((vA)->f[i] * (vC)->f[i]) + (vB)->f[i];
}

void _vmrghw( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB )
{
   VMX_reg  v;
   ulong  i, j;
   for ( i = 0; i < 2; i++ ) {
      j = i + i;
      v.ul[j] = (vA)->ul[i];
      v.ul[(j+1)] = (vB)->ul[i];
   }
   for ( i = 0; i < 4; i++ )
      (vT)->ul[i] = v.ul[i];
}

void _vmrglw( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB )
{
   VMX_reg  v;
   ulong  i, j;
   for ( i = 0; i < 2; i++ ) {
      j = i + i;
      v.ul[j] = (vA)->ul[(2+i)];
      v.ul[(j+1)] = (vB)->ul[(2+i)];
   }
   for ( i = 0; i < 4; i++ )
      (vT)->ul[i] = v.ul[i];
}

void _vmsubfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vC, VMX_reg *vB )
{
   ulong  i;
   for ( i = 0; i < 4; i++ )
      (vT)->f[i] = ((vA)->f[i] * (vC)->f[i]) - (vB)->f[i];
}

void _vnmsubfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vC, VMX_reg *vB )
{
   ulong  i;
   for ( i = 0; i < 4; i++ )
      (vT)->f[i] = -(((vA)->f[i] * (vC)->f[i]) - (vB)->f[i]);
}

void _vslw( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB )
{
   ulong  i, sh;
   for ( i = 0; i < 4; i++ ) {
      sh = (vB)->ul[i] & (ulong)0x1f;
      (vT)->ul[i] = (vA)->ul[i] << sh;
```

```
    }
}

void _vspltisw( VMX_reg *vT, long SIMM )
{
   ulong  i;
   for ( i = 0; i < 4; i++ )
      (vT)->l[i] = (long)(SIMM);
}

void _vspltw( VMX_reg *vT, VMX_reg *vB, ulong UIMM )
{
   ulong  i, ul;
   ul = (vB)->ul[(UIMM) & 0x3];
   for ( i = 0; i < 4; i++ )
      (vT)->ul[i] = ul;
}

void _vsubfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB )
{
   ulong  i;
   for ( i = 0; i < 4; i++ )
      (vT)->f[i] = (vA)->f[i] - (vB)->f[i];
}

void _vxor( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB )
{
   ulong  i;
   for ( i = 0; i < 4; i++ )
      (vT)->ul[i] = (vA)->ul[i] ^ (vB)->ul[i];
}
```

```
/*****************************************************************\
|*  File Name:     ppc_vmx.h                                    *|
|*  Description:   Header file for PPC vmx (altivec) emulation  *|
|*                                                              *|
|*            Mercury Computer Systems, Inc.                    *|
|*            Copyright (c) 1999  All rights reserved           *|
|*                                                              *|
|* Revision      Date        Engineer; Reason                   *|
|* --------      ----        ----------------                   *|
|*   0.0        991119         jg;  Created                     *|
\*****************************************************************/

#define  uchar   unsigned char
#define  ushort  unsigned short
#define  ulong   unsigned long

/*
 *  define a structure to represent a VMX (SIMD) register
 */
typedef union {
   char     c[16];
   uchar   uc[16];
   short    s[8];
   ushort  us[8];
   long     l[4];
   ulong   ul[4];
   float    f[4];
}  VMX_reg;

/*
 *  condition register comprised of 8 4-bit fields (0 - 7)
 */
extern  long  CR[];

/*
 *  prototypes for functions that emulate vmx instructions
 */
void _lvewx( VMX_reg *vT, ulong rA, ulong rB );
void _lvx( VMX_reg *vT, ulong rA, ulong rB );
void _stvewx( VMX_reg *vS, ulong rA, ulong rB );
void _stvx( VMX_reg *vS, ulong rA, ulong rB );
void _vaddfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB );
void _vmaddfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vC, VMX_reg *vB );
void _vmrghw( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB );
void _vmrglw( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB );
void _vmsubfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vC, VMX_reg *vB );
void _vnmsubfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vC, VMX_reg *vB );
void _vslw( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB );
void _vspltw( VMX_reg *vT, VMX_reg *vB, ulong UIMM );
void _vspltisw( VMX_reg *vT, long SIMM );
void _vsubfp( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB );
void _vxor( VMX_reg *vT, VMX_reg *vA, VMX_reg *vB );

/*
 *  vmx instuction macros
 */
```

```
#define LVEWX( vT, rA, rB )                 _lvewx( &vT, (ulong)rA, (ulong)rB );
#define LVX( vT, rA, rB )                   _lvx( &vT, (ulong)rA, (ulong)rB );
#define STVEWX( vS, rA, rB )                _stvewx( &vS, (ulong)rA, (ulong)rB );
#define STVX( vS, rA, rB )                  _stvx( &vS, (ulong)rA, (ulong)rB );
#define VADDFP( vT, vA, vB )                _vaddfp( &vT, &vA, &vB );
#define VMADDFP( vT, vA, vC, vB )           _vmaddfp( &vT, &vA, &vC, &vB );
#define VMRGHW( vT, vA, vB )                _vmrghw( &vT, &vA, &vB );
#define VMRGLW( vT, vA, vB )                _vmrglw( &vT, &vA, &vB );
#define VMSUBFP( vT, vA, vC, vB )           _vmsubfp( &vT, &vA, &vC, &vB );
#define VNMSUBFP( vT, vA, vC, vB )          _vnmsubfp( &vT, &vA, &vC, &vB );
#define VSLW( vT, vA, vB )                  _vslw( &vT, &vA, &vB );
#define VSPLTW( vT, vB, UIMM )              _vspltw( &vT, &vB, UIMM );
#define VSPLTISW( vT, SIMM )                _vspltisw( &vT, SIMM );
#define VSUBFP( vT, vA, vB )                _vsubfp( &vT, &vA, &vB );
#define VXOR( vT, vA, vB )                  _vxor( &vT, &vA, &vB );
```